PTC : Proxies that Transcode and Cache
in Heterogeneous Web Client Environments

Aameek Singh, Abhishek Trivedi, Krithi Ramamritham*
IIT Bombay

Prashant Shenoy
UMASS Amherst

Abstract

Advances in computing and communication technologies have resulted in a wide variety of networked mobile devices that access data over the Internet. In this paper, we argue that servers by themselves may not be able to handle this diversity in client characteristics and intermediate proxies should be employed to handle the mismatch between the server-supplied data and the client capabilities. Since existing proxies are primarily designed to handle traditional wired hosts, such proxy architectures will need to be enhanced to handle mobile devices. We propose such an enhanced proxy architecture that is capable of handling the heterogeneity in client needs—specifically the variations in client bandwidth and display capabilities. Our architecture combines transcoding (which is used to match the fidelity of the requested object to client capabilities) and caching (which is used to reduce the latency for accessing popular objects). Our proxies can intelligently adapt to prevailing system conditions using learning techniques to intelligently decide whether to transcode locally or fetch an appropriate version from the server. Our experimental results indicate that such strategies produce significant improvements in the client response times. Further, we find that even simple learning techniques can lead to significant performance improvements.

1. Introduction

The explosive growth of the World Wide Web has been accompanied by a proliferation of mobile devices with networking capabilities. Client devices differ significantly in their hardware characteristics (display resolutions, processing capacities), software capabilities and network connectivity. For instance, a typical networked PDA has 16MB memory, a 320x200 color display and 802.11b wireless interface, while a typical web-enabled phone has a black and white text-only display with a cellular data connection [5]. Further, the bandwidth of a typical cellular data connection is 9.6-19.2 Kbps (144 Kbps for GRPS), while that of a 802.11b connection is 11 Mbps.

Due to these differences, different versions of the same object may be suitable for different client devices. In case of images, for instance, a low resolution color version may be suitable for a small-screen color PDA, while a 2 gray bits/pixel version may be suitable for a black and white PDA. Similarly, in case of mobile phones with text-only displays, textual summaries of web pages may be more desirable than their full length counterpart. There are three possible techniques for handling such diverse client needs.

- Maintain all possible versions of the object at the server, one for each type of device.

- Store only the high fidelity version of the object and employ online transcoding to dynamically produce low fidelity versions [9].

- Employ an intermediate proxy that uses a combination of transcoding and caching to meet client needs.

Server-based techniques for managing the diversity in client needs have certain limitations. Maintaining multiple pre-computed versions of each object can be cumbersome, especially in scenarios where there is a large heterogeneity in the types of client devices. For example, many financial institutions maintain multiple versions of their web sites, one for traditional web clients, another for networked Palm PDAs, and yet another for mobile phones—each new device type adds to the overhead of maintaining and updating different versions. While the use of online transcoding for producing lower fidelity versions addresses this limitation, transcoding is known to be compute-intensive, and consequently, does not scale during periods of heavy loads and frequent updates. A more desirable approach is to offload the responsibility of handling different client types to a proxy. Such a proxy can fetch a high fidelity version of a requested object and use transcoding to produce a lower fidelity version. Further, it can cache the high-fidelity version, the transcoded version, or both, to quickly respond to future requests. Such a combination of transcoding and caching allows flexibility in how future requests are serviced—the proxy may respond to a request by (i)

sending a cached version (in the event the requested version is cached locally), (ii) transcoding a cached version to the desired fidelity (if a higher fidelity version is cached), (iii) sending a lower fidelity cached version than the one requested (during periods of heavy loads), or (iv) by downloading the object from the server.

Although the use of transcoding in web proxies has been investigated [5, 6], techniques for combining transcoding and caching techniques to reduce the overall resource usage at the proxy have not received much attention. Further, techniques to adapt the transcoding process to network conditions, proxy load and changing client requirements have not been explored. Our present work attempts to address these issues by making the proxy both intelligent and adaptive and by improving the effectiveness of transcoding by integrating it with caching mechanisms. We make three contributions in this paper.

- *Intelligent transcoding proxies*: Our proxy architecture is capable of learning and making appropriate policy decisions based on prevailing network and load conditions. We have implemented learning policies for adaptive proxies, which choose between server downloads and local transcodings based on the recent history of the particular client-server pair. Unlike prior approaches that choose between the two extremes of transcoding and no transcoding [6], our proxies can support a continuum of choices and use current system conditions to make an appropriate choice on a per request basis.

- *Adaptive Model*: Our adaptive model ensures that policy decisions are governed by the current prevailing conditions; further, the model chooses the least expensive option available to serve appropriate data to the client. We use simple techniques like linear regression and maintaining logs to estimate system conditions such as available bandwidth, proxy load, etc.

- *Transcoding-conscious Cache Replacement*: We combine caching and transcoding techniques by developing a cache replacement policy that takes into consideration transcoding utility of any cached object. This aspect of work is similar to that of [4]; the two approaches are contrasted in Section 5.

In what follows, we first describe, in Section 3, our system architecture for Proxies that Transcode and Cache (abbreviated as *PTCs*). We present our techniques for combining transcoding and caching in an intelligent fashion in Section 4 and examine the cache replacement problem in Section 5. Our focus in this paper is on transcoding and caching of web images. We experiment with image transcoding and caching with PTCs in Section 6. Section 7

concludes the paper with a discussion of future work.

2. System Architecture

The basic architecture of our Transcoding and Caching Proxy is shown in Fig-1. The main components of this architecture are:

1. **Client**: The client device can be a desktop, a laptop, a PDA or a mobile phone. Clients are assumed to piggyback their capabilities (in terms of the data fidelity they can handle) with each request; this specification can be included in the HTTP headers of the request [2]. Alternatively, the client capabilities can be specified in a client profile that is made available to the proxy. In addition to the client-specified characteristics, the proxy may additionally use the current system conditions to determine the fidelity appropriate for the client.

2. **Proxy**: Our proxy consists of a transcoding engine, a cache, and a resource manager. The resource manager is responsible for making decisions when multiple options are available to serve the data. These decisions are made based on our proposed policies, which have been detailed in the next section. The proxy takes a request from a client device, recognizes the capabilities of the client and accordingly fulfills the request. To do so, it might have to get the data from the *source* server and transcode it into the required version on-the-fly or serve the content from its local cache, possibly after transcoding it.

3. **Server**: The web server, *source* of the data, is enhanced in order to understand the requirements of the client and proxy and act accordingly. In the event multiple versions of the object are available (e.g., a text-only web page and a graphics-rich page), the source sends the proxy a version that has at least the required fidelity.
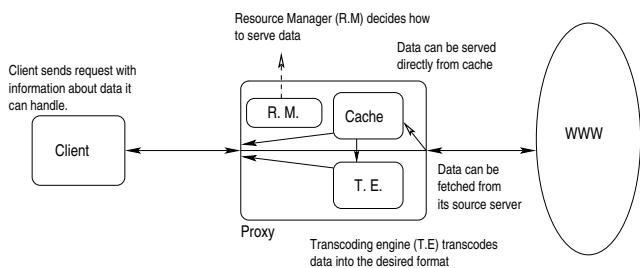
Figure 1. System Architecture: Transcoding and Caching Proxy

Our system works as follows.

- The **client** device sends a HTTP request to the proxy in the normal fashion except that it has some headers added to the request which contain the information about the content, the client is comfortable with. For example, it can have a header "*Quality*" indicating the quality of the object best suited to the client. For an image, this may represent the JPEG quality level, whereas for text this may represent the level of summarization that the client device desires, with higher quality meaning lesser summarization and more content.

- The **proxy** receives the request and identifies the client requirements. The proxy can use one of several strategies to service this request.

  - It may have the exact required version of the data in its local cache, in which case, it can just send that version to the client.

  - It may have some higher fidelity version of the data, which can be transcoded locally to the required fidelity.

  - If the server has the required version, then it may simply get that and send it to the client.

  - In case the server does not have the required version, then it may get a higher quality version and transcode is locally and send it back to the client.

  The exact strategy used by the proxy to effect this choice is detailed in the next section.

- The **server** receives a request from a proxy. That request may or may not contain a special "*Quality*" header. The presence of such a header indicates that a particular version of the data is being requested. If the server has that version, it will send it to the proxy. In case it does not have that version and the proxy desires, it will send the next higher version. Note that it may so happen that a proxy might have a higher version in the cache, in which case, it might request the server to just send the exact version, if present.

3. Transcode or Download?

Assuming that the proxy maintains a local cache, each client request results in one of four possibilities:

1. **Full Hit**: The cache contains the exact version requested by the client.

2. **Partial Hit**: This may occur, when the cache has a higher fidelity version. The higher version can be transcoded locally to the desired version For example, proxy has a quality 75 JPEG image when the request was for the quality 50 JPEG of the same image.

3. **Secondary Hit**: This happens when the proxy has a version with a lower fidelity than the one requested, proxy has the option of either downloading the appropriate version from the server, or serve the lower version, if that is acceptable to the client.

4. **Miss**: A miss occurs when there is no version of the data available in the cache.

Clearly, full hits and misses are easy to handle—the requested object is served using locally cached data in the event of a full hit or downloaded from the server in the event of a miss (if the server returns a higher fidelity version, a transcoding step is necessary before responding to the client). In the rest of this section, we outline our strategies to handle partial and secondary hits.

3.1. Partial Hit

As mentioned earlier, a partial hit is said to occur when the cache contains a higher fidelity version than requested by the client. Such a version can be easily transcoded to the desired version. This situation presents the following possibilities to the proxy:

- Transcode the higher version to the required version and send it to the client. (**Option T**)

- Check if the server has the required version and if it has, send that to the client after downloading from the server. (**Option D**)

If the server does not have the required version, we choose Option T. In case the server also has the required version, our proxy chooses one of the options trying to minimize the response time of the client. The decision is made after considering the following factors:

- **Complexity of transcoding**: This may in turn depend on the version of the object to be converted from (*before*) and the version to be converted into (*after*), the object type and size of that object [6].

- **Proxy Load**: For a dedicated proxy, the number of clients being served is a good estimate of the load.

- **Network Delays** between the source and proxy.

To decide between Option T and Option D, we propose the following:

1. Maintain the following statistics for each application of the transcoding process:

   - Size of the data before and after transcoding (s_b and s_a respectively).

- The number of clients connected to the proxy (n). This represents the current load on the dedicated proxy.

- The time taken to transcode from Version-A to Version-B (t_{AB}).

2. Also maintain the following information about any data that is downloaded by the proxy.

- Size of data (s_d).

- Source of data (i.e., the server, where it is downloaded from) (S).

- Time taken to download that data (t_d). This will be the round trip delay between the proxy and the server.

For a typical transcoding method and image type, as a first approximation, we assume that, t_{AB}, the time for transcoding is a function of its object size (s_b) and the number of clients connected to the proxy (n).

$$t_{AB} = \mathcal{T}_{AB}(s_b, n)$$

where $\mathcal{T}_{AB}$ is some function. Now, if we can approximate the function $\mathcal{T}_{AB}$, we can approximately calculate the time that transcoding a version A to version B would take under particular load conditions on the proxy. This is where the statistics come into play. We store all the above mentioned information and when we have enough data, we evaluate t_{AB} beforehand by approximating $\mathcal{T}_{AB}$. An important point to notice is that the prediction models may be different for different kinds of objects (for example, JPEG-JPEG transcoding generally varies linearly with s_b, while it is not so for JPEG-GIF transcoding [6]).

On similar lines, we try to predict, t_d, the time to download a particular object from its source (*Option D*). There has been considerable amount of work done in this area [7, 10]. As an approximation, we assume the download-delay to be a function of the size of the object to be downloaded and the proxy load (represented by the number of clients being served by the proxy).

$$t_d = \mathcal{D}_S(s_d, n)$$

Then we attempt to approximate $\mathcal{D}_S$. A point to be noted here is that we do not have to interact with the server to get the size of the data to be downloaded (s_d). Since we have a higher version of that data, s_d can be estimated by using the transcoding data, s_b (size before transcoding) and s_a (size after transcoding) values in particular, for that transcoding and other data format information [6].

This estimate would give us the time that would be taken by the proxy to download the requested version of the object under present conditions (*Option D*). However, a simple comparison between t_{AB} and t_d might not be appropriate.

It is usually preferable to serve clients with local data primarily because network behavior might be erratic and hence difficult to capture using a simple prediction model. So we impose a condition that if t_{AB} is greater than M times t_d then it is better to download from the server, where M is a tunable parameter. Thus, the heuristic is of the form:

If $t_{AB} \leq M * t_d$
then transcode locally
else download the required version from the source.

The parameter M can be tuned to achieve different transcoding/download ratios depending upon the prevailing conditions other then estimated times to transcode/download. For example, in case the proxy is heavily loaded, M can be decreased to cause more server downloads. Similarly in case of light proxy load conditions, the value of M can be increased, causing more local transcodings.

There might be times when we don't have enough data to estimate T and D. In such cases we set the default to be the local transcoding of the higher version present in the cache.

Clearly, we need ways to approximate functions $\mathcal{T}_{AB}$ and $\mathcal{D}_S$. For this, we propose a number of estimation techniques, ranging from simple and basic techniques to complex algorithms, which give better approximations but result in processing overheads which cannot be ignored. A few sample techniques are:

1. **IP based**: A simple scheme which maintains a record of IP addresses for servers, and chooses between local transcoding and downloading from the server depending on the source address. It learns to make a decision, depending upon a single factor of server address. On a very basic level, this can divide the servers into local servers and remote servers, and can learn to choose local transcoding for remote servers and downloading from the servers if they are local. But clearly this would be prone to changes in network behavior for a single client-server pair.

2. **Min-Min Comparison**: Another possibility is for a scheme which assumes a best case scenario for both *Option T* and *Option D*, that is, it assumes that the next request for a similar sized file would take the minimum of the times taken by both the options in the last N transcodings or server downloads. This scheme is fairly simple and does not use the statistics of the load on the proxy, bandwidth available, etc. However, since we are using only the very recent times as the basis of approximations, the decision is based on the prevailing conditions and takes into account the available bandwidth, load on proxy and server, etc.

3. **Multiple Linear Regression**: A little more complex policy is that of using *Multiple linear regression*. It is a

common learning technique for linear models. We assume $\mathcal{T}_{AB}$ and $\mathcal{D}_S$ to be linear functions and learn this function using regression. For this, we use the statistics for load on the CPU (represented by the number of client connections), the time taken for transcodings and server downloads, and the size of the file to predict the time to transcode locally, and the time to download the required version from the server and make an appropriate decision.

The techniques which we have used are fairly simple ones, and employ the past performance to predict times to transcode and download. Another option is to use a more complicated policy for *Option T* as mentioned in [6]. The more accurate policy has to be based on a variety of other factors, like the content of the image (that is whether the image is a natural image or an artificially rendered image), image dimensions, compression algorithm and transcoding parameters (depth of quantization and/or scaling) [6]. Such a policy would more accurately predict t_{AB}, but would be computationally demanding as well.

3.2. Secondary Hit

Secondary hit is said to occur when the cache at the proxy has a lower fidelity version than requested. In this case there are the following possibilities:

- If the user can compromise on QoS or it is known that this version has all the data that might be of interest to the user, we can respond with the lower fidelity version.

- If the server has the required version, the proxy gets that version from the server and sends it to the user.

- If the server does not have the required version, the proxy gets a higher version and transcodes it to the required version.

In case of a secondary hit the decision has to be made keeping in mind the tradeoff between time to get a higher version from the server, and compromise on quality by the client. This should be done on the basis of predefined course of action for the server-client pair, with inputs like available bandwidth, and predicted time to download from the server. Such a scenario can be dealt with by integrating a personalization engine with the proxy, so that we have the exact details of user interests and his or her agreement to compromise on QoS.

4. Cache Replacement Policies

Any cached object can be accessed for two reasons:

- The proxy gets a request, specifically for that object. We call this, a **Direct Reference**.

- The proxy gets a request for a lower version of an object and it has to transcode this object to the desired version. We call this, **Transcoding Reference**.

Each object stored at a proxy will hence have two kinds of utilities:

- **Reference Utility**: This is the utility of the object because of the time it saves when a client requests this object (that is, a Full Hit occurs), since this object is directly sent to the client from the local cache. This is similar to the normal scenario of a cache hit.

- **Transcoding Utility**: This is the utility of the object, because of the time it saves when a client requests a lower version of this object (that is, a Partial Hit occurs). The object can now be transcoded to the desired version and sent to the client.

Notice that it might be profitable to keep an object in the cache which is not referenced directly even once (very low reference utility) but is extremely useful since lots of requests are being served by transcoding it to other requested versions (very high transcoding utility)!

To accommodate such a scenario, we assign a profit metric to each cached object. This profit metric is an augmented version of the metric used in WATCHMAN [8]. Each cached object, O_i, has a profit value, P_i, given by:

$$P_i = \frac{\lambda_i c_i}{s_i} + \frac{\sum_j \gamma_{ij} b_{ij}}{s_i}$$

j: All possible versions into which O_i can be transcoded.
λ_i: Average rate of direct reference of the object.
γ_{ij}: Average rate of reference of the object, when it is referenced for transcoding to version j.
c_i: Cost saved by the presence of this object, when it is referenced directly.
b_{ij}: Cost saved by the presence of this object, when it is referenced for transcoding to a version j.
s_i: Size of the object.

In the above expression, $\lambda_i.c_i$ corresponds to the reference utility of the object. It determines the cost savings due to caching O_i for direct reference requests. In case two objects have the same cost savings, it is beneficial to evict the one with the larger size as it frees more space in the cache. Hence the inverse relationship with s_i.

The second subexpression corresponds to the total transcoding utility of the object. The expression, $\gamma_{ij} \, b_{ij}$ corresponds to the transcoding utility of the cached object for its transcoding to version j. Since an object can be transcoded to a number of versions, we need to sum this over all possible j's. Again the s_i factor appears because of the preference to evict larger objects.

4.1. Calculation of parameters

In this section, we discuss techniques to determine λ_i, γ_{ij}, c_i and b_{ij}. First let us see how we determine λ_i and γ_{ij}. λ_i is estimated based on a moving average of the last K inter-arrival times of requests to O_i. Notice that these references are direct references. It is defined to be:

$$\lambda_i = \frac{K}{t - t_K}$$

where t is the current time and t_K is the time of the last K^{th} reference. The inclusion of current time t in the expression ensures that the objects which were referenced long back have lesser profit values. This guarantees aging of cached objects. In case less than K references are available, λ is estimated using the available references. Similarly, γ_{ij} is estimated based on the moving average of last K inter-arrival times of accesses to O_i for transcoding to Version j.

$$\gamma_{ij} = \frac{K}{t - t_K}$$

Now we need to determine c_i and b_{ij}. Let us assume that there are n versions of an object in the cache, $\{ V_1 \dots V_n \}$ with V_1 being the lowest version and V_n being the highest version.

Notice the fact that V_1 can be generated in two ways:

- Download from its source (Time taken for this option is given by t_{s_1})

- Transcode from V_j, $j = min(2, ..., n)$ where V_j is in cache. (Time for this option is given by $t_{V_j \to V_1}$)

We take the minimum over j for the second option, since it is computationally cheaper to transcode an object from an immediate next higher version, which will have smaller size than other available versions. Now, the proxy will generate V_1 by the method which takes minimum time. Therefore,

$$c_1 = min(t_{s_1}, t_{V_j \to V_1}) , j = min(2, ..., n) \text{ such that } V_j \text{ is in cache.}$$

Now V_1 will have zero transcoding utility since it cannot be transcoded into any other version. Therefore,

$$b_{1j} = 0 \ \forall \ j$$

Let us consider a version V_i where $i \in \{2, ..., n-1\}$. On similar lines,

$$c_i = min(t_{s_i}, t_{V_j \to V_i}), j = min(i+1, ..., n) \text{ such that } V_j \text{ is in cache.}$$

Now, let us consider its transcoding utility. It can be used to transcode into $i - 1$ versions. Therefore, whenever it is accessed for transcoding to Version j, it saves the amount of time saved thereby is given by the minimum of the time to download Version j from its source or to transcode it from a version higher than i. Hence,

$$b_{ij} = min(t_{s_j} - t_{V_i \to V_j}, t_{V_k \to V_j} - t_{V_i \to V_j}),$$
$$k = min(i+1, ..., n) \text{ such that } V_k \text{ is in cache.}$$

For the highest version V_n, it is easy to see,

$$c_n = t_{s_n}$$
$$b_{nj} = t_{s_j} - t_{V_n \to V_j}$$

4.2. Maintenance of parameters

With each cache object, we maintain statistics for its last K direct and K transcoding references for each type of transcoding from this version to another version. Whenever the object is accessed, we update these statistics, depending on whether it was a direct or a transcoding reference. In order to avoid overheads of updating λ and γ for changing values of t, profit values are evaluated on-the-fly whenever the object has to be considered for eviction from the cache.

The other parameters requiring regular updates are c_i and b_{ij}. c_i and b_{ij} depend on the presence of other versions of the objects in the cache as well. Hence, we will have to update these values whenever a new version of the same object is either brought in or evicted out of the cache. These periodic updates help since these values also depend on the present load conditions on the proxy and it will make sure that they stay true to the current conditions.

4.3. Algorithm

Since the objects with lesser references have less reliable estimates of λ_i and γ_{ij}, the cache replacement algorithm gives them a higher priority for eviction. As [8] suggests, we consider all objects with just one reference (Direct and Transcoding) and evict the ones with least profit scores. Then we consider the objects with two references and so on. The parameters used are (i) S: Size of object to be cached, (ii) C: Set of objects to be replaced, and (iii) Max: Maximum number of references to any object.

Algorithm 1 Replace(S)

for all $i = 1$ *to Max* **do**
    R_i = *list of retrieved set of objects with exactly i references arranged in increasing profit order*
end for
R = *list of all retrieved set of objects arranged in order*
$R_1 < R_2 < \cdots < R_{Max}$
C = *minimal prefix of R such that* $\sum_{O_j \in C} s_j \geq S$
return C

4.4. Performance

We implemented the above mentioned Cache Replacement Strategy and compared it with LRU. The data objects accessed consisted of 300 JPEG images of 3 different versions. The total size of the images is 100 Mb. The access

trace was created using Zipf's law [3] with $\alpha = 0.5$. There were a total of 1000 requests by 5 concurrent clients. Fig-2 contains the plot of hit ratio with different cache sizes. The hit ratio is defined to be $\frac{\#Full\ Hits + \#Partial\ Hits}{\#Total\ Requests}$. As the plot indicates, our algorithm (PTC) works very well as compared to standard LRU. This is because of the replacement policy being based simply on the reference and transcoding utilities of the object instead of being based on the last access to the object.
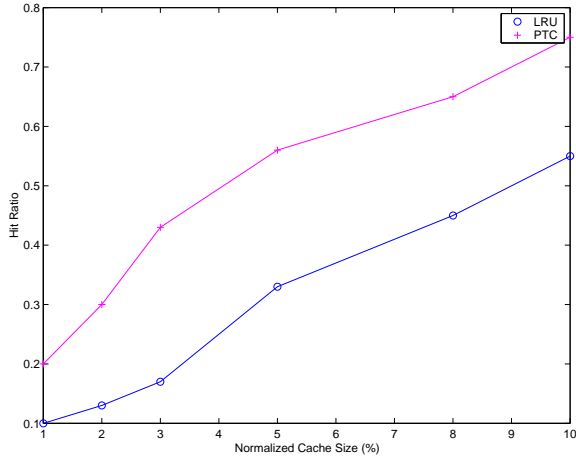
Figure 2. Cache Replacement

We would like to point out that a technique similar to the above, for transcoding-conscious cache replacement, was recently proposed in [4]. The work uses the notion of a weighted transcoding graph and adopts a general profit function based on the aggregate effect of each object. For each cached object, a corresponding weighted transcoding graph is maintained. For example, consider a graph in Fig-3. The nodes indicate various versions for a cached object and the edges represent the transcoding relationships between the various versions. For example, an edge between node 1 and node 2 indicates that version 1 can be transcoded into version 2 with transcoding cost represented by the weight of the edge (10). A generalized profit function is formulated in a manner similar to the one devised in our work.

Despite the similarities, there are three main differences between the two:

- They assume constant *transcoding complexity* for a specific *before* and *after* transcoding pair. In general, system conditions such as the current proxy load can significantly impact this value; our work takes such variations into account.

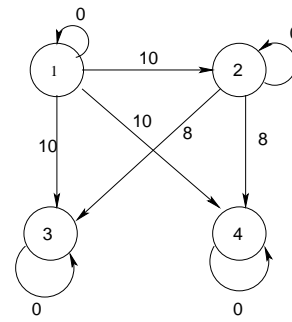- They assume a constant server download rate from the server. In general, network conditions on the server-

Figure 3. Weighted Transcoding Graph

proxy path may vary depending on the network traffic, an aspect that is considered in our work.

- Finally, we use learning approaches to endow intelligence to our proxies and learn from past history—an important difference that can potentially yield performance improvements.

5. Experimental Evaluation of PTCs

5.1. Setup

Our system consists of a proxy, an enhanced web server and a number of client devices. The web *server* is a modified version of Apache-1.3.19. The experiments used a collection of a three thousand JPG images, which were collected from local users at IIT Bombay. The server has 3 versions of each file of quality factors 100, 50 and 20. The *proxy* is a modified Java-based transcoding proxy called Rabbit [1]. The transcoding mechanism used is the "*convert*" utility in Linux. The *client* devices are simulated by Perl scripts which connect to the proxy and simulate HTTP requests containing special "*Quality*" headers. All the three components are on regular desktop machines with Pentium III, 800 MHz processors and 128 MB of RAM. Two different sources, local and remote, are used in order to do a controlled performance evaluation of our proposed policies. The local source is a machine connected to the proxy via IIT Bombay LAN. The remote source is located at the University of Massachusetts, Amherst. We implemented 4 different policies and compared their total response times.

5.2. Partial Hit Policy

Recall that a partial hit occurs when the cache contains a higher version of the data requested and the source has the exact requested version of that data. In such a scenario, the proxy has to choose between downloading the data from its source server, or transcoding the higher version locally.

To show the performance improvement when we introduce intelligence in proxies, we compare two intelligent

polices for PTCs, *PTC:Min-Min* and *PTC:Regression*, and compare their performance with a third policy, *Download-all*, which always chooses server download, and a fourth, *Transcode-all*, which always chooses local transcoding.

Regression policy approximates the time to transcode and time to download from the server on the basis of number of connections, and the size of files. It is implemented with a learning time of 16 connections, i.e., it starts to take decisions for partial hits after 16 partial hits have taken place. During the first 16 connections, it always chooses local transcoding.

Min-Min policy is implemented with $N = 5$, i.e., it approximates the time for local transcoding or for server download from the last 5 connections of that data size range, for that particular client-server pair. The data items are divided in groups of 10 KB in our implementation. Because the files are grouped in size ranges, to make an approximation, at least 5 partial hits must have happened in that particular size range.

5.2.1. Traces. Requests for 3000 files were made by 5 concurrent clients. The trace was created using Zipf's law [3] with $\alpha = 0.5$. Each request for a 100 quality factor image is followed by requests for a lower version of the same image. There were around 1200 partial hits for this request stream. The set of experiments were run with both local and remote sources.

5.2.2. Performance. Fig-4 shows the plot of total response time for the stream of requests for eight different runs for a local source, with $M = 4$. Such a situation simulates the scenario when the source-proxy communication is fast.

Not surprisingly, since the server is local, in almost all the runs the download-all option has the minimum response time. It is very closely followed by the PTC policies. Transcode-all has the maximum response time. That the PTC policies are performing well is encouraging given their processing overheads and the effect of the initial learning period.

The performance of the PTC policies lies in its ability to recognize the fast server-proxy link and hence doing minimum amount of local transcoding. This is clearly evident from Fig-5, which plots the number of times, PTC policies chose to transcode a higher version locally. For Regression policy, there are close to 16 local transcodings in all runs, which is the number of transcodings in the learning phase. The Min-Min policy has a larger number of transcodings due to longer learning phase, and hence the total response time is also more.

A similar set of experiments were run with a remote source. Fig-6 plots the total response times for the four approaches for this case (with $M = 4$).
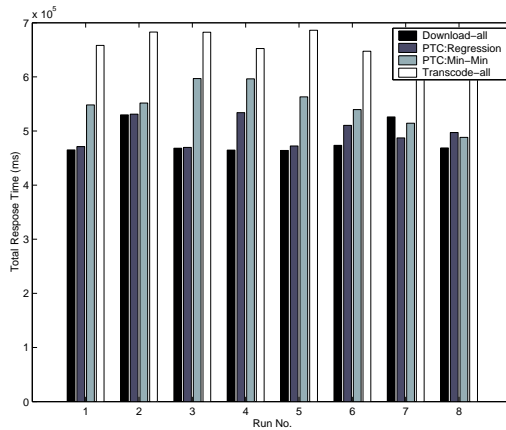
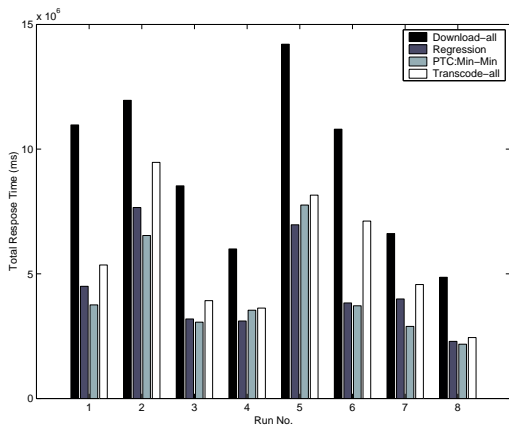**Figure 4. Total Response Times (Local Source; M=4)**
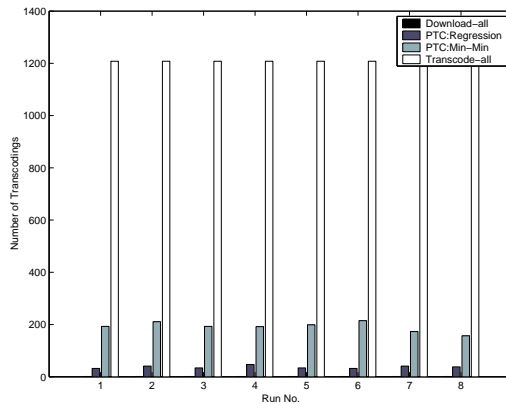
Figure 5. No of transcodings (Local Source; M=4) Note: In download-all case the no. of local transcodings is 0

As the plot indicates, the PTC policies turns out to be the near best in almost all cases. This again is because of its ability to choose the better of the two available options upon the occurrence of a partial hit. Fig-7 plots the number of times local transcoding was preferred over server download. Clearly, with a remote server, local transcoding is chosen in more than half the cases, since it is better to transcode the object locally than getting it from a remote source.

5.2.3. Analysis of Performance.

To better understand the performance of the policy, we narrowed down the performance for *full hits*, *partial hits* and *misses*. We plotted the performance of two policies (Regression and Min-Min) for the cases of a local and a remote server. Fig-8 plots the response times for the local server. The only non-intuitive part is a greater response time for a
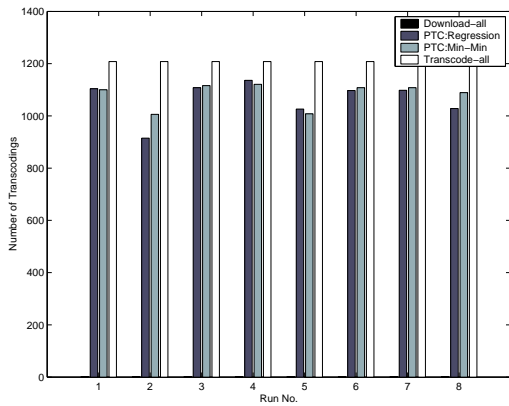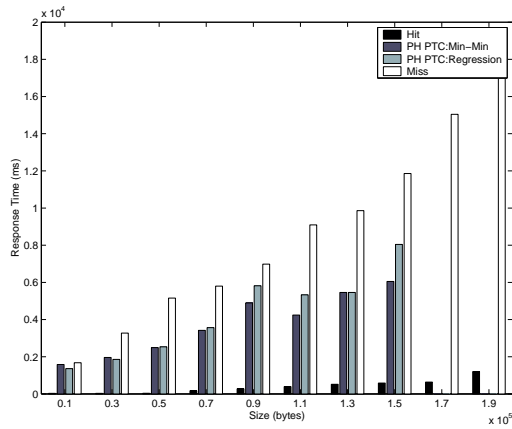
Figure 6. Total Response Times (Remote Source; M=4)
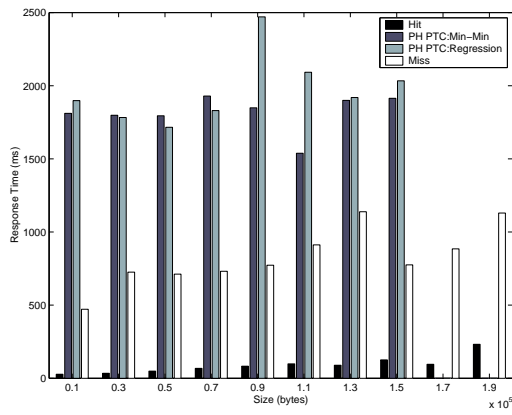
Figure 8. Breakup of performance (Local Server) Note: There are no partial hits for higher data sizes

Figure 7. No of transcodings (Remote Source; M=4) Note: In download-all case the no. of local transcodings is 0

Figure 9. Breakup of performance (Remote Server)

partial hit than a miss for a similar sized data object. This is explained by the fact that in case of a local server, the policy mostly chooses to download the object from the server for a partial hit. Hence the time would at least be equal to the time of a miss. The difference is attributed to the processing overheads for the policy, which include the learning phase.

Fig-9 contains a similar plot for a remote server. Clearly the partial hit policy is reducing the total response time. Since in most of the real-life scenarios, the web server is remote, the use of such a policy is justified and beneficial.

As seen in the last two plots, there is very little to choose from the two policies even though one of the policies (Regression) is more accurate than the other (Min-Min). This indicates that the processing overheads are significant and are influencing the total response times. This is further clear from Fig-10.

The regression policy is taking around 20% of the total response time, whereas a simpler Min-Min policy is taking just 5%. Hence, we claim that a more accurate and complex policy would not necessarily better the performance and even a simple policy can be good enough.

5.2.4. Performance for Different Request Ratios. To compare the performance of our adaptive policies we calculate the total response time for different request sets, with the percentage of local requests varying from 0 to 100%.

As we see in Fig-11, the PTC policies, both Min-Min and Regression perform far better than the policies which always choose local transcoding over server download or vice versa, for almost all request ratios. Only when 95% or more of the requests are local, is there not much to choose
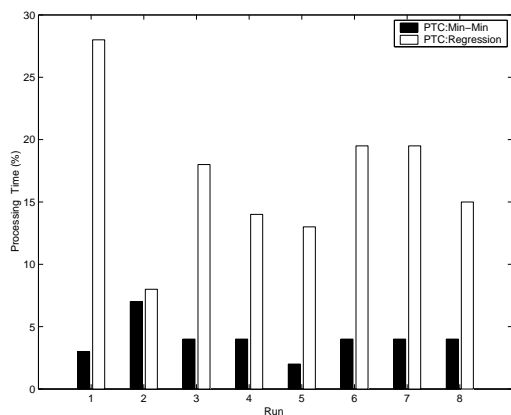
Figure 10. Percentage of response times spent in processing (Local Server)
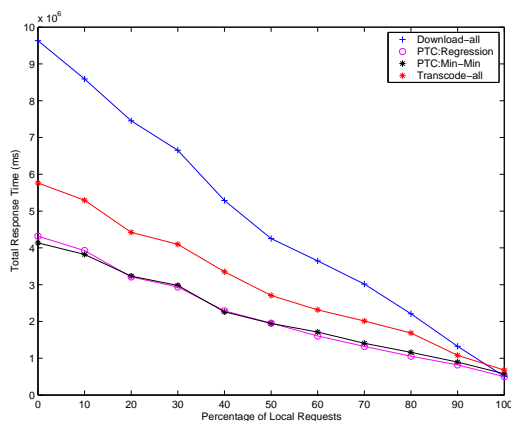
Figure 11. Response Times for Different Proportion of Local Requests

between the policies. Since this situation is highly unlikely, simple, yet intelligent, policies are highly desirable.

6. Conclusions and Future Work

In this work, we have developed intelligent proxies to efficiently handle diverse client devices accessing the WWW. The proxies have been integrated with a caching engine and the resulting caching issues have been identified and strategies proposed to handle these issues. As our results indicate, our adaptive policies make the most appropriate choice. Also, the proxy adapts itself to the current network traffic and proxy load conditions. We have shown that basic and simple policies can lead to considerable performance improvements. We have also developed a cache replacement algorithm which has been shown to work much better than vanilla LRU.

This work can be extended to include more proxy-level

contribution to the Partial Hit policies, for example, the proxy can act on its own depending upon the prevailing conditions, and choose an appropriate version for the client. We need to develop and implement a personalization engine, which keeps profiles for clients, and helps make better decisions for secondary and partial hits. For example, in case a client with a small-screen handheld wants to see only a selected portion of the data, we can use this personalized information to send just the clippings the user is interested in. Also, another extension would be broadening the scope of data objects by including various kinds of data like videos, voice, text etc. There is also a need for more extensive testing of various prediction mechanisms, especially with objects that require different types of transcoding.

References

[1] http://rabbit-proxy.sourceforge.net/.
[2] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakyl. An active transcoding proxy to support mobile web access. *The 17th IEEE Symposium on Reliable Distributed Systems*, 1998.
[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the implications of zipf's law for web caching. Technical Report CS-TR-1998-1371, 1998.
[4] C. Y. Chang and M. S. Chen. Exploring aggregate effect with weighted transcoding graphs for efficient cache replacement in transcoding proxies. *In Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE-02)*, 2002.
[5] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. *In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, Massachusetts*, pages 160–170, 1996.
[6] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile WWW browsing. *IEEE Personal Communication*, 5(6):8–17, 1998.
[7] S. B. Moon, J. Kurose, and D. Towsley. Packet audio playout delay adjustment: Performance bounds and algorithms. *Multimedia Systems*, 6:17–28, 1998.
[8] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN : A data warehouse intelligent cache manager. In *The VLDB Journal*, pages 51–62, 1996.
[9] C. S. E. Surender Chandra and A. Vahdat. Differentiated multimedia web services using quality aware transcoding. In *INFOCOMM*, 2000.
[10] T. Trump. Estimation of clock skew in telephony over packet switched networks. *In Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, 1:2605–2608, 2000.